Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/6040--11-9357

TURBID: A Routine for Generating Random Turbulent Inflow Data

LEE PHILLIPS
DAVID FYFE

Laboratory for Propulsion, Energetic, and Dynamic Systems
Laboratories for Computational Physics and Fluid Dynamics

November 9, 2011

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
09-11-2011	Memorandum Report	

4. TITLE AND SUBTITLE

TURBID: A Routine for Generating Random Turbulent Inflow Data

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Lee Phillips* and David Fyfe

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
64-4492-0-1

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory (Code 6042)
4555 Overlook Avenue, SW
Washington, DC 20375-5344

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/6040--11-9357

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR / MONITOR'S ACRONYM(S)

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

*Work performed at NRL, currently employed at Alogus Research Corporation, 1434 Waggaman Circle, McLean, VA 22101.

14. ABSTRACT

We report on a Fortran module for the generation of turbulent inflow and initial data for use in fluid simulation codes. We explain the uses and limitations of the module and how to incorporate it into an existing code. The data created by the module is examined, and the code is tested for parallel performance on shared memory, multi-processor computers; a complete listing is included.

15. SUBJECT TERMS

Turbulence Fluid simulation
OpenMP Turbulent boundary conditions

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					David Fyfe
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	Unclassified Unlimited	21	**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified			(202) 767-5863

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

This page intentionally left blank.

TURBID: A Routine for Generating Random Turbulent Inflow Data

1 Introduction

Simulations of turbulent or fluctuating fluid flows require, in the case of simulation domains with closed boundaries, initial conditions to be defined throughout the fluid volume, and, in the case of simulation domains with one or more inflow boundaries, the fluid to be fully specified at the inlet. These initial and inflow boundary conditions are in addition to the boundary conditions (rigid wall, free-slip surface, compliant membrane, etc.) that must be specified on the remaining boundaries of the domain for a correctly posed mathematical problem.

In any system of partial differential equations, either their analytical solution or a numerical approximation to it depends critically upon the boundary and initial conditions. In the case of time-dependent, evolving, turbulent flows with an inflow boundary, which will be the subject of this report, the gross features of the evolving flow field strongly depend on the details of the inflow. In fact, even the large-scale mean features, their time evolution, and the character of the eventual statistically steady state, if one is attained, depend on the detailed statistical character of the fluid at the inflow boundary.

Manuscript approved October 24, 2011.

An example of this possibly counterintuitive result was given by Klein *et al.* in 2002. [1] These authors display solutions of a plane, turbulent jet problem for three different types of inflow conditions: a specified mean flow with no fluctuations, the mean flow with added random, uniformly distributed (in wavenumber space) fluctuations, as might be calculated by simply adding the scaled output of a random number generator to an appropriate mean flow profile, and a more physically realistic inflow taken as the output of a separate direct numerical simulation of turbulent flow in a channel. The interesting result was that, after a short transient interval, the first two cases become identical; the inflow field containing uniform, random fluctuations led to the same result as the inflow consisting of a perfectly laminar mean field with no fluctuations. Distinct from these solutions is what is presumably the "correct" one, resulting from a more realistic inflow condition.

The reason [1] the uniform inflow and the inflow with uniformly random fluctuations lead to nearly identical, and incorrect, results is that the energy distribution in the fluctuations of the latter field are not physically realistic for the problem at hand (or for any problem involving real fluid turbulence). The uniform distribution puts too much energy at high wavenumbers, which are quickly damped in the solution and lead to a result insensibly different from the purely laminar case. The uniform distribution of fluctuations contains no information about the length scales relevant to the problem, and is not sufficiently physically realistic.

In principal, one could solve the problem of generating physically realistic inflow conditions by evolving an initially laminar flow in the domain desired, by direct numerical simulation, through its transition to turbulence, and using the result as a boundary condition. However, this would likely be a bigger project, and consume more computational resources, than the main problem supposedly under consideration, and is not normally considered practical.

The problem of generating a sufficiently realistic inflow field using a method that is efficient enough for inline use in simulation codes has been attacked from several angles, and several good solutions have appeared in the literature. We report here on the implementation of one of these solutions in a general-purpose routine in the form of a FORTRAN module, TURBID, that can easily be compiled into any fluid simulation code. We examine the output of the module, give examples of its use in the generation of inflow and/or initial volume data, and test its scalability under shared-memory parallelism.

2 The code

The Appendix contains a complete listing of the source, with comments retained. The language is FORTRAN90 with openMP [2] directives (discussed below). Although the code is designed to be general-purpose and should be easily plugged in to any existing code, certain considerations relating to current projects which are likely to make use of TURBID led to several design decisions. The code is written as a FORTRAN module and uses no common blocks. Certain arrays are allocated but not released, as the routine will usually be called repeatedly, and can reuse the associated memory. If the code is used only to initialize a volume and will not be called again during the run, the user may wish to modify it to deallocate the arrays.

TURBID is a straightforward implementation of the procedure and algorithm described by Klein (2002). [1] It produces random but *spatially correlated* velocity fields, with a Gaussian correlation function. The technique uses a digital filter and remains in physical space throughout, and is inherently fairly efficient. The dominant length scales, which can be different in each coordinate direction, are set as parameters `lsx, lsy, lsz` in terms of the grid spacing. The user will assign physical dimensions to the grid cells and timestep as appropriate to the problem.

The Gaussian nature of the correlations is physically faithful to the nature of three-dimensional homogeneous turbulence in its most elementary form. The results [1] calculated using such inflow data show that it retains enough physical reality to lead to correct solutions in a variety of turbulence problems. In contrast, more naively constructed random inflow data, such as that incorporating uniform, random fluctuations, leads to incorrect results. [1] In short, the fields calculated by this routine probably contain the minimum essential physical ingredients to lead to realistic simulation results that can be expected to agree with experiments, provided that strong departures from Gaussian statistics are not a prominent feature of the problem under investigation. Such non-Gaussian statistics may occur in strongly intermittent [3] or two-dimensional turbulence [4]. In such cases somewhat more elaborate methods [5] for generating synthetic fluctuations may be required.

The main function in the module, `turin`, returns the velocity field to the calling program. The geometry is Cartesian (x,y,x), with x in the direction of the mean flow, and the inflow boundary in the $(x-y)$ plane. The velocity field is stored as a structure with components `velx`, `vely`, and `velz`, defined at the beginning of the module. `turin` calls `vfromr`, which calculates the velocities from uniform random number fields, which are returned form the standard random number

generator. `vfromr` in turn uses the function `filco`, which calculates the filter coefficients. The correlation function is defined in `ffunc`; the user can replace the Gaussian here by any desired correlation function without needing to change anything else in the code; in this way other distribution functions, such as Gaussians with long tails that are associated with intermittent turbulence, can be tried.

After the module is supplied a main program; its chief purpose is to illustrate the use of the module, especially the required declarations, and to call it to generate test output and timings. The main program would normally be deleted before the code is used in production. The user will normally scale the field of fluctuations returned by `turin` to reflect the desired turbulence intensity, and add the fluctuations to a particular mean flow profile.

The user of the routine has several parameters to set to control operation of the code. All of these are passed as arguments to the call to `turin`, which is normally the only external interface intended to be used. The first two arguments to `turin` are the dimensions of the grid perpendicular to the mean flow; the routine does not need to know about the streamwise extent of the computational box. After that comes `iseed`, for initializing the random number generator. Repeating the run with the same `iseed` will yield the same results. The next three arguments, `lsx, lsy,` and `lsz` are the scale lengths in units of the grid cell size. There are three because it is possible to have different scale lengths in each direction. The filter widths, `nfx, nfy,` and `nfz`, can normally be set to twice the scale lengths, as is done in the example main program. The final argument is a boolean flag that tells the routine whether to initialize a new inflow plane or to calculate a new inflow plane that is correlated in the x (or temporal) direction with the one calculated in the previous call. A study of the supplied main program should make clear how this parameter is used.

3 Results

The main program produces four output files. The $x, y,$ and z components of velocity on the entire grid are stored in `fort.9, fort.10,` and `fort.11`, respectively; the data is appended to these files for each iteration. Another file, `tseries`, records all components of the velocity field at one specified grid location for each iteration. It can be used to examine the single-point spectra of the output, for example, considered as a time series.

The grid data can be plotted with a routine like the following `python` program:

```
from numpy import *
from pylab import *
interactive(False)
ny = nz = 100
vx = loadtxt('fort.9')
for f in range(len(vx)/nz):
   pcolormesh(arange(ny), arange(nz), vx[f*nz:f*nz+nz-1, :],
              shading="flat")
   savefig('vxplot%i.png' % f)
vy = loadtxt('fort.10')
for f in range(len(vx)/nz):
   pcolormesh(arange(ny), arange(nz), vy[f*nz:f*nz+nz-1, :],
              shading="flat")
   savefig('vyplot%i.png' % f)
vz = loadtxt('fort.11')
for f in range(len(vx)/nz):
   pcolormesh(arange(ny), arange(nz), vz[f*nz:f*nz+nz-1, :],
              shading="flat")
   savefig('vzplot%i.png' % f)
```

This code saves an image of each velocity component for each iteration. The images can be assembled into a movie, which provides a view of the data as a time-varying inflow boundary. An example of such as movie can be seen at `http://lcp.nrl.navy.mil/\~lphillip/ct/turbulentInflow.mov`, which was calculated using a length scale of `lsx = 6`.

Another view of the same data could be provided by assembling the frames into a volume by stacking them at equal intervals along the x axis. This would represent the simulation volume initialized with random data. The main program contains a commented-out loop introduced with a comment that shows how to use the module to fill arrays holding the velocity components in case the entire simulation volume needs to be initialized with a turbulent field. The module can be used for such initialization alone, to generate inflow turbulence only, or for both.

Figure 1 shows an $x-y$ plane for one velocity component for three values of the length scale. The velocities range from -1 to 1 (red to blue); the user must scale the fluctuations to represent the desired level of turbulence in the simulation. The first frame shows the result when the length scale is set to one grid cell, which

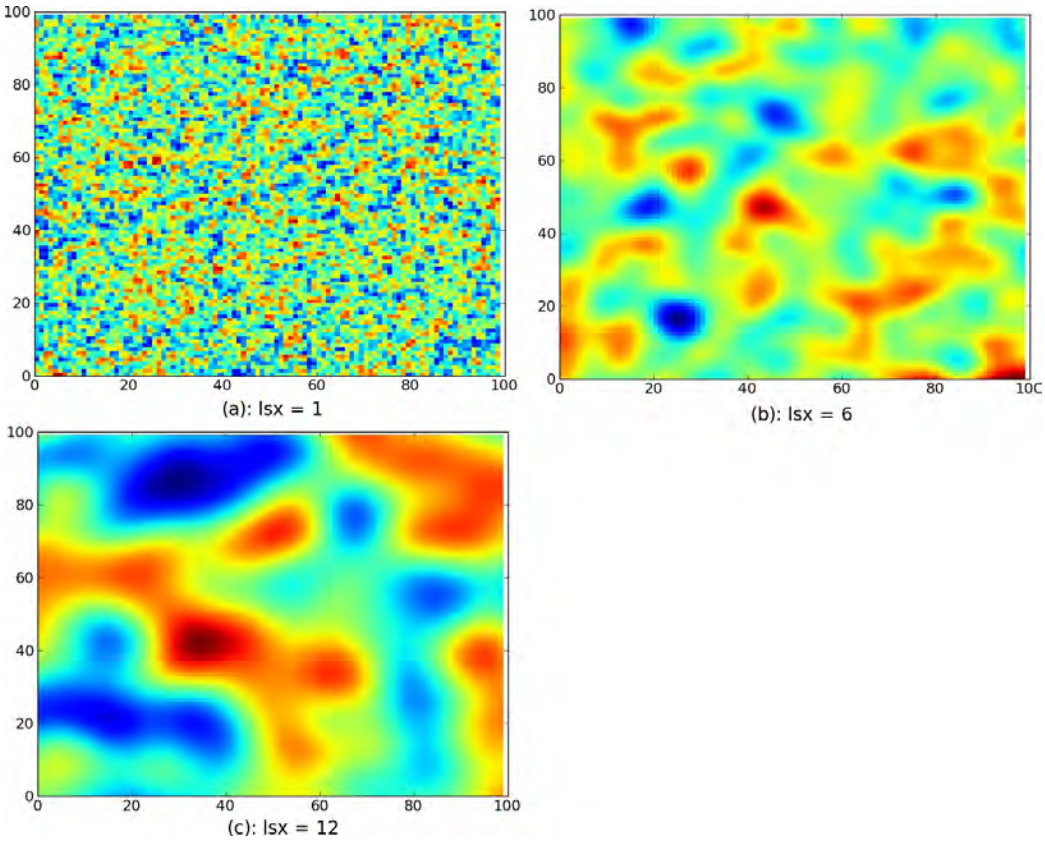

Figure 1: One component of the turbulence field for three values of the length scale `lsx`. The data plotted ranges from -1 to 1; frame (a) is equivalent to uniformly random, uncorrelated data.

is equivalent to uniformly random, uncorrelated data.

It should be noted here that the calculated fluctuations contain no real physics. The routine does not know the equations of motion or anything about real flows, except for the correlation function defined in `ffunc` and the length scales passed in to the routine by the user. It simply returns a set of random numbers filtered in such a way that they are correlated according to the correlation function over the specified length scales. We typically use a Gaussian correlation function because that agrees with measurements of fluid turbulence in some class of experiments and with some models of turbulent fluctuations. But the fluctuations returned by the routine should not be expected to agree with, for example, the Kolmogorov spectrum, which arises from physical processes that are unknown to the routine.

In a sense this is desirable, as it allows the generated boundary conditions to be used in a variety of situations where there are minimum assumptions made about the physics or the nature of the medium. The distribution of fluctuations calculated is "statistically correct" in the sense that it leads to correct results when used as an inflow condition in a simulation code. [1] The correct turbulent spectrum will be developed downstream of the inflow by the fluid code, which will contain the actual physics of dissipative and nonlinear processes. The fluctuations calculated by this routine also do not satisfy the incompressibility condition, and so may equally be used in compressible and incompressible solvers. In the case of an incompressible flow problem, the fluctuations will add an error in the form of a small (when compared with the incompressible mean flow) nonzero $\nabla \cdot \mathbf{v}$ term. In a compressible simulation at low Mach number this may create a small perturbation in the form of low-amplitude sound waves that will propagate away. In the case of an incompressible simulation, the error can be removed as part of the normal solution procedure by using projection methods [6] or other means; in any case the $\nabla \cdot \mathbf{v}$ error is not more than 1% of the streamwise velocity in typical cases. [7]

One check that the results reflect the desired statistical correlations and are therefore, at least to that degree, correct, can be made by calculating the two-point autocorrelation, $\mathscr{C}(\tau) = \int v(x)v(x+\tau)\,dx$ of the output fluctuations. This was done by taking one example 100×100 (y,z) plane from the code output with all length scales set to 12, and extracting several lineouts, calculating the 1-d $\mathscr{C}$ for each lineout. The results are plotted as solid lines in Figure 2, along with Gaussian curves reflecting the same length scale, $e^{-((x-50)/12)^2)}$, as dotted lines.

In interpreting the figure one should not pay too much attention to the areas near the boundaries at 0 and 100, as the results there are polluted by the usual edge effects that occur when numerically estimating autocorrelations. We can see that, although the results vary statistically depending on exactly where the lineout is taken, all the autocorrelation curves reflect fairly accurately the desired Gaussian structure, and they have the correct scale length. Each Gaussian curve in the plot was normalized to have the same central amplitude as an autocorrelation curve to facilitate comparison. The results are a good indication of the correctness of the code, especially considering the noisiness of the statistics and numerical estimation of the correlation that can be expected on the small grid.

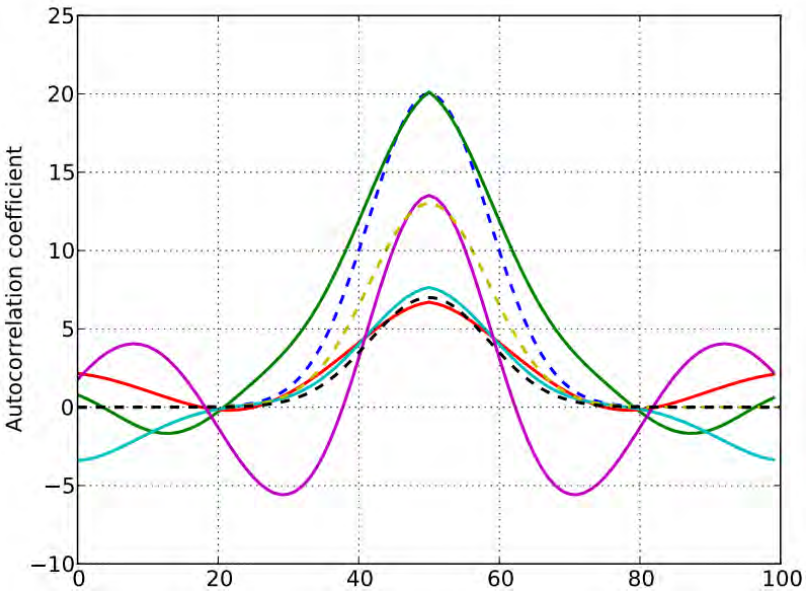

Figure 2: Autocorrelation of turbulent fluctuation field (solid lines) compared with Gaussians at the target scale length (dotted lines).

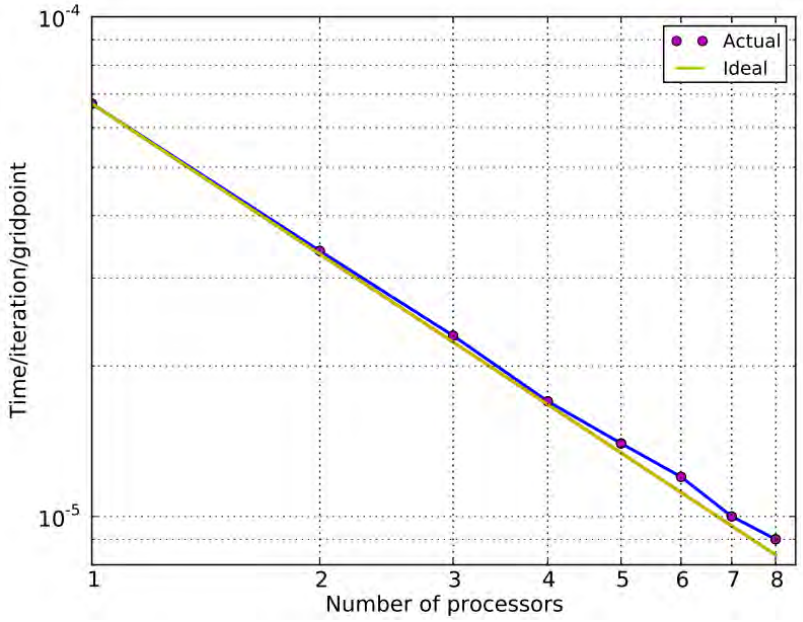

Figure 3: The runtime *vs* number of processors, showing nearly perfect parallel scaling.

4 Runtimes

Figure 3 shows the time in seconds per complete iteration per gridcell as a function of N, where N is the number of threads, or processors, used. The runs were performed on a machine with eight processors each sharing main memory, so N runs from 1 to 8. The values plotted are each an average of five runs on a 100×100 grid, with each run consisting of 100 iterations of the routine. The scale lengths were all set to 6 and the filter widths were set to their normal values of twice the scale lengths.

The values are plotted on a log-log scale and compared with the ideal $1/N$ scaling. It is clear that there is nearly perfect parallel scaling. This parallel scaling arises entirely from several openMP [2] directives, which implement shared-memory parallelism. An examination of the source code shows that we have only

applied the openMP directives to one set of loops in the `vfromr` function. Various other parts of the code were subjected to parallelization experiments, but this did not lead to any further speedup. In particular, use of `OMP WORKSHARE` [8] directives applied to various array-syntax statements found in the code worsened performance on the machine used for testing. We verified that the output from the routine was not affected by the number of processors used.

Further work in increasing the parallel efficiency of the program might involve rewriting some of the array-syntax statements as explicit loops which may be more amenable to openMP optimizations and, for incorporation in codes using distributed memory architectures, organizing the code along MPI-type lines.

Appendix: TURBID source listing

The code is FORTRAN90 using fixed line format, and contains openMP directives as well as conditionally compiled timing statements. It is available online at `http://lcp.nrl.navy.mil/ lphillip/ct/turbid.F`. If the file is saved with the extension `.F`, then it can be compiled using `ifort` with the command `ifort -openmp turbid.F`. This will create a module that can be used in another code by following the example given in the `main` program, which is the last routine in the listing.

In order to run timing tests, add the argument `-Dtiming` to the compilation command. This turns off all writing to files except for the timing result, which is written to the file `timing` after the timing results are collected.

```
module inflow
type :: velvec
   real :: velx
   real :: vely
   real :: velz
end type
real, allocatable :: bijk(:,:,:)
! This module allocates several arrays and does not deallocate
! them. It is intended to be called frequently and reuses the
! arrays. If you call the routines here only at startup you may
! want to deallocate storage.
contains
```

```
      function turin(ny, nz, iseed, lsx, lsy, lsz,
     +                nfx, nfy, nfz, startup)
         ! If startup then initialize full random number fields:
         ! we are starting a new problem; if not startup then
         ! shift random fields and calculate temporally and
         ! spatially correlated data.
         ! lsx and lsy are the length scales as multiples
         !       of gridcells.
         ! The filter widths nfx, nfy, and nfz should be at least
         ! twice the length scales. Normally nfy = nfz.
         logical startup
         integer lsx, lsy, lsz, ny, nz
         integer nfx, nfy, nfz ! filter widths
         ! The random fields:
         real, dimension(-nfy+1 : ny+nfy, -nfz+1 : nz + nfz) :: rfill
         real, allocatable, save :: rx(:, :, :), ry(:, :, :),
     +                              rz(:, :, :)
         ! The vector velocity, for easy return from this function:
         type(velvec), dimension(ny,nz) :: velvecs, turin
         integer iseed
         integer k,j,ip,jp
         ! x is the mean flow direction; y is 'vertical'; z is
         ! crosswise
         if (startup) then
            allocate(rx(-nfx : nfx, -nfy+1 : ny+nfy, -nfz+1 :
     +                  nz + nfz))
            allocate(ry(-nfx : nfx, -nfy+1 : ny+nfy, -nfz+1 :
     +                  nz + nfz))
            allocate(rz(-nfx : nfx, -nfy+1 : ny+nfy, -nfz+1 :
     +                  nz + nfz))
            call random_seed(iseed)
            call random_number(rx)
            call random_number(ry)
            call random_number(rz)
            rx = 2. * rx - 1.
            ry = 2. * ry - 1.
            rz = 2. * rz - 1.
         else
```

```
               ! Shift random fields:
               call random_number(rfill)
               rfill = 2. * rfill - 1.
               rx = eoshift(rx, 1, rfill, 1)
               call random_number(rfill)
               rfill = 2. * rfill - 1.
               ry = eoshift(ry, 1, rfill, 1)
               call random_number(rfill)
               rfill = 2. * rfill - 1.
               rz = eoshift(rz, 1, rfill, 1)
            end if
            velvecs = vfromr(rx, ry, rz, nfx, nfy, nfz, ny, nz, lsx,
     +                       lsy, lsz)
            turin = velvecs
         end function turin
         function vfromr(rx, ry, rz, nfx, nfy, nfz, ny, nz, lsx, lsy,
     +                   lsz)
            type(velvec), dimension(ny,nz) :: velvecs, vfromr
            real, dimension(-nfx : nfx, -nfy+1 : ny+nfy, -nfz+1 :
     +                      nz + nfz) :: rx, ry, rz
            integer lsx, lsy, lsz, ny, nz, nfx, nfy, nfz
            integer k,j,ip,jp
            if (.not. allocated(bijk)) then
               allocate(bijk(-nfx:nfx, -nfy:nfy, -nfz:nfz))
            endif
            bijk = filco(lsx, lsy, lsz, nfy, nfz)
            velvecs%velx = 0.
            velvecs%vely = 0.
            velvecs%velz = 0.
!\$OMP PARALLEL private(k, j, kp, jp, ip)
!\$OMP DO
            do k = 1, nz
               do j = 1, ny
                  do kp = -nfz, nfz
                     do jp = -nfy, nfy
                        do ip = -nfx, nfx
                           velvecs(j,k)%velx = velvecs(j,k)%velx +
     +                     bijk(ip,jp,kp) * rx(ip, j+jp, k+kp)
```

```
                        enddo
                     enddo
                  enddo
               enddo
            enddo
!\$OMP END DO
!\$OMP DO
            do k = 1, nz !Nasty repeated code; but too much abstraction
               do j = 1, ny ! here would make this hard to follow.
                  do kp = -nfz, nfz
                     do jp = -nfy, nfy
                        do ip = -nfx, nfx
                           velvecs(j,k)%vely = velvecs(j,k)%vely +
     +                     bijk(ip,jp,kp) * ry(ip, j+jp, k+kp)
                        enddo
                     enddo
                  enddo
               enddo
            enddo
!\$OMP END DO
!\$OMP DO
            do k = 1, nz
               do j = 1, ny
                  do kp = -nfz, nfz
                     do jp = -nfy, nfy
                        do ip = -nfx, nfx
                           velvecs(j,k)%velz = velvecs(j,k)%velz +
     +                     bijk(ip,jp,kp) * rz(ip, j+jp, k+kp)
                        enddo
                     enddo
                  enddo
               enddo
            enddo
!\$OMP END DO
!\$OMP END PARALLEL
            vfromr = velvecs
         end function vfromr
         function ffunc(k, n)
```

```
   ! The actual filtering function. Notice that it's a
   ! Gaussian.
   real ffunc
   real pi
   integer k, n
   parameter (pi = 3.1415926)
   ffunc = exp(-(pi*k**2 / (2.*n**2)))
end function ffunc
function filco(lsx, lsy, lsz, nfy, nfz)
   !Returns the array of filter coefficients.
   ! lsx, etc. are the length scales;
   ! nfy, etc. are the filter widths.
   integer lsx, lsy
   integer nfx, nfy, nfz
   real s
   real, allocatable :: filco(:,:,:), bx(:), by(:), bz(:)
   nfx = nfz
   allocate(filco(-nfx:nfx, -nfy:nfy, -nfz:nfz))
   !The 1d coefficients that are multiplied together to form
   !the filco array:
   allocate(bx(-nfx:nfx), by(-nfy:nfy), bz(-nfz:nfz))
   s = 0.
   do k = -nfx, nfx
     s = s + ffunc(k, lsx)
   enddo
   s = sqrt(s)
   do k = -nfx, nfx
     bx(k) = ffunc(k, lsx) / s
   enddo
   s = 0.
   do k = -nfy, nfy
     s = s + ffunc(k, lsy)
   enddo
   s = sqrt(s)
   do k = -nfy, nfy
     by(k) = ffunc(k, lsy) / s
   enddo
   s = 0.
```

```
            do k = -nfz, nfz
              s = s + ffunc(k, lsz)
            enddo
            s = sqrt(s)
            do k = -nfz, nfz
              bz(k) = ffunc(k, lsz) / s
            enddo
            do k = -nfz, nfz
               do j = -nfy, nfy
                  do i = -nfx, nfx
                     filco(i,j,k) = bx(i)*by(j)*bz(k)
                  enddo
               enddo
            enddo
            deallocate(bx, by, bz)
         end function filco
      end module inflow
      program main
         use inflow
         integer    iseed
         data         iseed/10023459/
         integer ii
         integer nx, ny, nz ! Grid size
         parameter (nx = 100, ny = 100, nz = 100)
         ! Vector velocity at the inflow plane:
         type(velvec), dimension(ny,nz) :: velvecs
         ! type(velvec), dimension(nx,ny,nz) :: v
         real, dimension(nx, ny, nz) :: vx, vy, vz
         integer lsx, lsy, lsz ! length scales as multiples of cell
         parameter (lsx = 12, lsy = 12, lsz = 12)
         integer nfx, nfy, nfz ! filter widths
         parameter (nfy = 2 * lsy, nfz = 2 * lsz, nfx = nfz)
         character (len=40) vformat
         character (len = 30) cnx
         integer iters
         parameter (iters = 1)
#ifdef timing
         integer :: time1, time0, tdelta
```

```
         real :: ttime
         open(Unit = 90, file="timing", form = "formatted", status =
     +    "new")
#endif
         write(cnx, *) nx
         vformat = '(' // cnx // '(g12.3))'
         ! Initialize the inflow plane:
         velvecs = turin(ny, nz, iseed, lsx, lsy, lsz,
     +                   nfx, nfy, nfz, .true.)
         ! This is how to fill the volume with velocity fluctuations:
c         do ii = 1, nx
c            velvecs = turin(ny, nz, iseed, lsx, lsy, lsz,
c     +                      nfx, nfy, nfz, .false.)
c            vx(ii, :, :) = velvecs%velx
c            vy(ii, :, :) = velvecs%vely
c            vz(ii, :, :) = velvecs%velz
c         enddo
         open(Unit = 80, file="tseries", form = "formatted", status =
     +    "new")
           write(80, *) 0 ! Header
#ifdef timing
         CALL SYSTEM_CLOCK(COUNT_RATE=tdelta)
         CALL SYSTEM_CLOCK(COUNT=time0)
#endif
         do ii = 1, iters ! Get a series of correlated inflow plane
            ! Write out "time series" at a particular location:
#ifndef timing
            write(80, *) ii, velvecs(1,1)%velx,velvecs(1,1)%vely,
     +                    velvecs(1,1)%velz
            write(9, vformat) velvecs%velx         ! fluctuations
            write(10, vformat) velvecs%vely
            write(11, vformat) velvecs%velz
#endif
            velvecs = turin(ny, nz, iseed, lsx, lsy, lsz,
     +                      nfx, nfy, nfz, .false.)
         enddo
#ifdef timing
         CALL SYSTEM_CLOCK(COUNT=time1)
```

```
         ttime = real( (time1-time0)/tdelta )
         write(90,*) ttime/(iters * nz * ny)
#endif
      end
```

Acknowledgments

This work was supported by the U.S. Naval Research Laboratory. We thank Gopal Patnaik for advice and support.

References

[1] M. Klein, A. Sadiki, and J. Janicka. A digital filter based generation of inflow data for spatially developing direct numerical or large eddy simulations. *J. Comp. Phys*, 186, 2002.

[2] Openmp.org. `http://openmp.org/wp/`.

[3] François N. Frenkiel and Philip S. Klebanoff. Statistical properties of velocity derivatives in a turbulent field. *J. Fluid Mech.*, 48, 1971.

[4] G. Boffetta, A. Celani, and M. Vergassola. Inverse energy cascade in two-dimensional turbulence: Deviations from gaussian behavior. *Phys. Rev. E*, 61, January 2000.

[5] L. di Mare et al. Synthetic turbulence inflow conditions for large-eddy simulation. *Phys. Fluids*, 18, February 2006.

[6] John B. Bell, Phillip Colella, and Harland M. Glaz. A second-order projection method for the incompressible navier-stokes equations. *Journal of Computational Physics*, 85:257 – 283, 1989.

[7] Andreas Kempf, Markus Klein, and Johannes Janicka. Efficient generation of initial- and inflow-conditions for transient turbulent flows in arbitrary geometries. *Flow, Turbulence and Combustion*, 74, 2005.

[8] Miguel Hermanns. Parallel programming in fortran 95 using openmp. http://www.openmp.org/presentations/miguel/F95_OpenMPv1_v2.pdf, 2002.